Data Structures and Algorithms

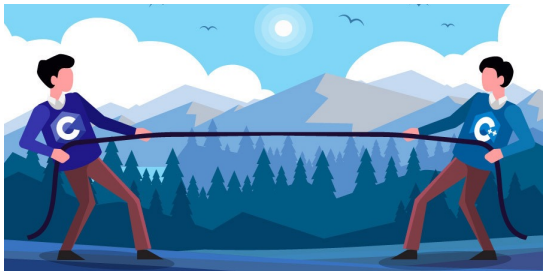
Chapter 2
Procedural Programming in C++

Brief Historic Note on C++

- C++ reads like «see plus plus»
- Created in 1983 by Bjarne Stroustrup as an extension of C, a.k.a. "C with classes"
- One of the most successful object oriented (OO) programming languages; OO programming has become very popular since the 1990s
- Standardized by ISO
  - Initial standard: ISO/IEC 14882:1998
  - Latest version: ISO/IEC 14882:2017, a.k.a. C++11

Main Differences between C++ and C

- The main one is the C++ support of classes! Later on...
- As a procedural language, C++ is mainly different in:
  - Comments
  - I/O from keyboard/console
  - String manipulation
  - Input/Output from/to Files
  - Dynamic memory allocation
  - Reference variable NEW
  - Function receiving or returning a reference NEW
  - Function overloading and default arguments NEW
  - Namespaces NEW



Same Simple Program in C and C++


```c
#include <stdio.h>

int main() {
 int n;
 /* This is a very simple program */
 printf("Hello World!\n");
 printf("Input an integer->");
 scanf("%d", &n);
 printf("Next one is %d\n", n+1);
 return 0;
}
```

```cpp
#include <iostream>
using namespace std;
int main() {
 int n;
 // This is a very simple program
 cout << "Hello World!" << endl;
 cout << "Input an integer->";
 cin >> n;
 cout << "Next one is " << n+1 << endl;
 return 0;
}
```

Comments in C++

- Comments are explanatory statements that are included within the code to document it and make it more readable
- Two types of comments in C++:
  - *Multi-line comment*: all characters inside any block delimited with `/*` and `*/`, containing an arbitrary number of lines, are ignored

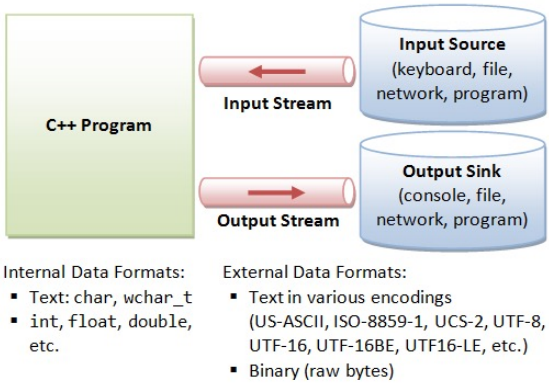
```
/* This is a comment */
```

```
/* Comments can also
   span multiple lines
*/
```

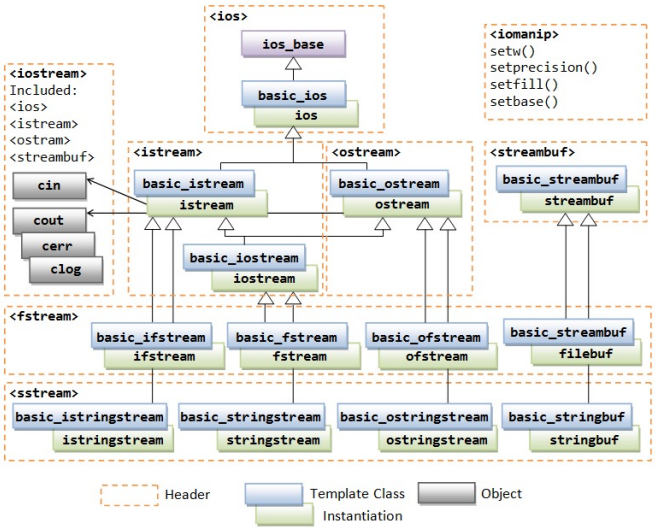
  - Single-line comment: all characters after `//` and *within the same line* are ignored (only available in C++)

```
double price = 100.0;
price *= 1.23; // add value-added tax (VAT) to the price
```


Input/Output in C++: Streams



- Stream: sequence of bytes flowing in or out the program
- `<iostream>`, `<fstream>`, and `<sstream>` libraries



- `<iomanip>` library for formatting

Input/Output from/to Keyboard/Console

- `#include <iostream>` : to use standard I/O streams
- Most common standard I/O streams and operators:
  - `cout` and the *insertion operator* <<

```
cout << value;
cout << value1 << value2 << ... ;
```

  - `cin` and the *extraction operator* >>

```
cin >> value;
cin >> value1 >> value2 >> ... ;
```


Data Structures and Algorithms
Estruturas de Dados e Algoritmos
2. Procedural Programming in C++
1 2 9 0 FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE D COIMBRA
DEPARTAMENTO DE ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

Examples of cout

```
int i = 1;
cout << i; // display the value of i
```

```
int i = 1, j = 2, k = 3;
double a = 0.0, b = -1.25, c = 1.3e4;

// build and display a stream with the value of multiple variables
cout << i << j << a << k << b << c;

// build and display more complex streams including strings
// the manipulator endl causes a newline character to be output
cout << "i=" << i << ", b=" << b << ", k=" << k << endl;
```

Examples of `cin`

- For these examples, assume the input stream: `12  17.3  -19`

```
int i, j;
double a;
cin >> i;  // i <-- 12
cin >> a;  // a <-- 17.3
cin >> j;  // j <-- -19
```

```
int i, j;
double a;
cin >> i >> a >> j;
// i <-- 12, a <-- 17.3, j <-- -19
```

```
int i, j, k, l;
char  c;
cin >> i >> j >> c >> k >> l;
// i <-- 12, j <-- 17, c <-- '.', k <-- 3, l <-- -19
```

  - The operator >> is "smart" enough to consider the type of the target variable when it determines how much to read from the input stream

Extraction Operator and Whitespace

- *Whitespace*: characters that do not produce a visible image
- Most common whitespace:

Character	Code
newline	\n
tab	\t
blank	(space)
carriage return	\r

- By default, the operator >> ignores whitespace characters
  - It removes *leading* whitespace characters from the input stream and discard them
  - The `get()` function (and others) can be used to read whitespace characters

Discard Characters from Input Stream

- `cin.ignore(N, ch)`
  - Skip (read and discard) up to `N` characters in the input stream, or until the character `ch` has been read and discarded, whichever comes first
  - Examples:

`cin.ignore(80,'\n');` ⟹ Skip the next 80 input characters, or skip characters until a newline char. is read, whichever comes first

`cin.ignore();` ⟹ Skip only the next input character

User Prompts with `cout` and `cin`

- Prompts: users must be given a cue when and what they need to input

```
int userAge;
cout << "Enter your age: ";
cin >> userAge;
```

- Because of buffering, it is possible that the prompt may not appear on a monitor before the `cin` is exec.
  - Solution: use manipulator flush

```
int userAge;
cout << "Enter your age: " << flush;
cin >> userAge;
```

Formatting the Output in C++

- `#include <iomanip>` : to format output streams
- `setw(width)`
  - Sets the field `width`, i.e. the number of characters in which the field is displayed
  - The setting applies to the next single value only

```
double a = 3.75;
cout << setw(6) << a << '|' << a '|';
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		3	.	7	5	|	3	.	7	5	|			

Formatting the Output in C++

- `setfill(ch)`
  - Specifies the character used to fill in the unused space in an output field when using `setw()`
  - If omitted, the pad character is the space (blank) character
  - The setting applies to all values after the manipulator

```
double a = 3.75;
cout << setfill('0') << setw(6) << a << '|';
cout << setw(10) << a << '|';
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	3	.	7	5	|	0	0	0	0	0	0	3	.	7	5	|		

Formatting the Output in C++

- `setprecision(digits)`
  - Sets the number of digits shown after the decimal point
  - Applies to all floating point values after the manipulator
- `fixed`
  - Force real values whose decimal part is zero to be printed with trailing zeros in fixed-point notation

```
double pi = 3.141592653589793;
cout << setw(9) << setprecision(3) << fixed << pi << '|';
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
				3	.	1	4	2	|					

Data Structures and Algorithms
Estruturas de Dados e Algoritmos
2. Procedural Programming in C++
1 2 9 0 FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE D COIMBRA
DEPARTAMENTO DE ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

Formatting the Output in C++

- `left, right`
  - Sets the justification to the `left` or to the `right`, respectively
  - The default alignment is to the `right`

```
double p[2] = {8.87373, 19.045123677};
cout << left << setw(15) << "Name" << right << setw(6) << "Grade" << endl;
cout << left << setw(15) << "James"
        << right << setw(6) << setprecision(2) << fixed << p[0] << endl;
cout << left << setw(15) << "Catherine"
        << right << setw(6) << setprecision(2) << fixed << p[1] << endl;
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
N	a	m	e													G	r	a	d	e				
J	a	m	e	s													8	.	8	7				
C	a	t	h	e	r	i	n	e								1	9	.	0	5				

Formatting the Output in C++

- Other manipulators:

Manipulator	Description
`scientific`	Show floating point values in scientific notation
`showpoint`	Always show decimal point for floating point values
`hex`	Display integer values in hexadecimal
`oct`	Display integer values in octal
`showbase`	Show default prefix for the base being used
`uppercase`	Present digits A-F and base prefix in uppercase
`showpos`	Precede every non-negative value with plus signal

Reading to Input Failure

- The stream variable returns an indication of success (true or false) when we attempt to extract a value
- Typical design logic in pseudocode:

```
Try to read data (often denoted as priming read)
While the last attempt to read data succeeded do
      Process the last data that was read
      Try to read data
Endwhile
```

- Alternative to using the stream variable to check:
  - `!<input_stream>.fail()` , e.g. `!cin.fail()`

Reading to Input Failure

```
const int MINPERHOUR = 60, MAXTIMES = 10;
int time[MAXTIMES]; // time value in total minutes
int hours, minutes; // HH field of time, and MM field of time
int numTimes = 0;   // number of time values read
int totalTime = 0;  // sum of all time values
cin >> time[0];     // priming read
while (cin && numTimes < MAXTIMES){// or: (!cin.fail() && ...
    totalTime += time[numTimes++];
    if (numTimes < MAXTIMES) cin >> time[numTimes];
}
cin.clear(); // reset the status flag after an input failure
for (int i = 0; i < numTimes; i++){
    hours = time[i] / MINPERHOUR;
    minutes = time[i] % MINPERHOUR;
    cout << setw(3) << i+1 << '|';
    cout << setw(3) << hours << ':';
    cout << setw(2) << setfill('0')
         << minutes << setfill(' ') << endl;
}
cout << "Total: " << totalTime << endl;
```

1	2	3	4	5	6	7	8	9	10
2	1	7							
4	9								
1	1	0							
3	0	2							
a									
		1	|			3	:	3	7
		2	|			0	:	4	9
		3	|			1	:	5	0
		4	|			5	:	0	2
T	o	t	a	l	:		6	7	8

C++ Strings

- C strings are essentially arrays of `char` delimited with `'\0'`…
- In C++, the `string` type can be used with many advantages
- `#include <string>` : to use C++ strings
- Declaration, assignment, concatenation, substring, access:

```
string name;                                    // declaration
name = "James";                                 // assignment
string surname = "Bond";       // declaration and assignment
string fullName = name + " " + surname;         // concatenation
cout << fullName.substr(6, 3);                  // display "Bon"
char initialSurname = fullName[6];              // 'B'
initialSurname = fullName.at(6);                // safer!
unsigned int len = fullName.length();           // len <- 10
```

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
J	a	m	e	s		B	o	n	d

C++ Strings

- More about declaration, assignment, access:

```
string s1 = "string1";                  // s1 <- "string1"
string s2("string2");                   // s2 <- "string2"
string s3(s2);                          // s3 <- "string2"
s1 = s1 + s2;                           // s1 <- "string1string2"
s1[6] = 'X';                            // s1 <- "stringXstring2"
s1[s1.length() - 1] = '.';              // s1 <- "stringXstring."
s1[s1.length()] = '!';                  // illegal index; it's ignored
```

- Comparison: ==, !=, <, <=, >, >=


```
string s1 = "Joe Dalton", s2("Jack Dalton");
string s3 = "William Dalton", s4("Averell Dalton");
bool a = (s1 == s2);   // a=false
bool b = (s1.substr(4, 6) != s2.substr(5, 6));      // b=false
bool c = (s4 < s1), d = (s3 >= s4);                 // c=true, d=true
```


C++ Strings

- Reading a C++ string *not containing space characters*:

```
string firstName;
cin >> firstName;
```

  - `cin` cannot be used to read space characters to a string because: i) it reads and discards from the input stream *leading whitespaces*; ii) it does not read and keep *trailing whitespace characters* in the input stream

- Reading a C++ string *containing space characters*:

```
string fullName;
getline(cin, fullName);
```

  - Read *every* character until new line (`'\n'`) or end of file (`EOF`) are detected

- Reading C strings:

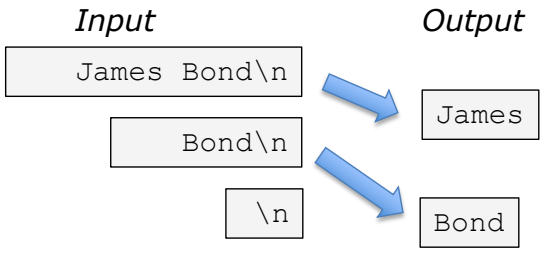
```
char firstName[15];
cin >> firstName;
```

```
char fullName[256];
cin.getline(fullName, 256);
```

C++ Strings

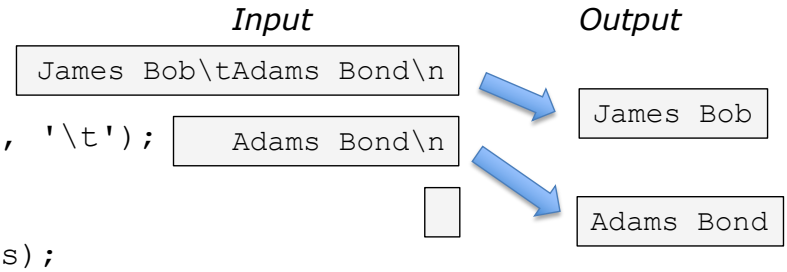
- Example 1:

```
string firstName;
cin >> firstName;
cout << firstName;
string other;
cin >> other;
cout << other;
```



- Example 2:

```
string givenNames;
getline(cin, givenNames, '\t');
cout << givenNames;
string familyNames;
getline(cin, familyNames);
cout << familyNames;
```

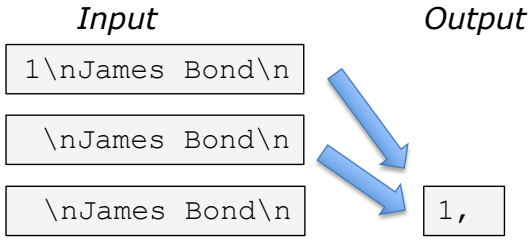


Data Structures and Algorithms
Estruturas de Dados e Algoritmos
2. Procedural Programming in C++
1 2 9 0 FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE D COIMBRA
DEPARTAMENTO DE ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

C++ Strings

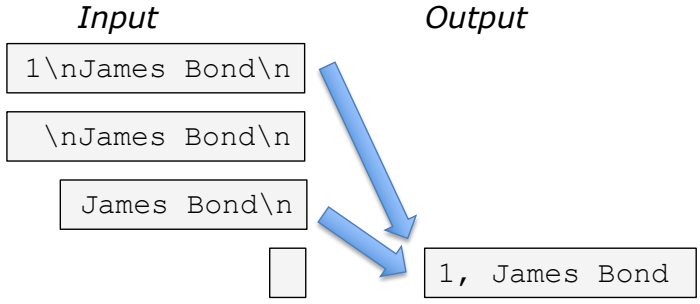
- Example 3:

```
int id;
cin >> id;
string name;
getline(cin, name);
cout << id << ", " << name;
```



- Example 4:

```
int id;
cin >> id;
cin.ignore();
string name;
getline(cin, name);
cout << id << ", " << name;
```



Convert C String to C++ String & v.v.

- C string to C++ string

```
char c_string[] = "James Bond";
// no need to worry about storage space :-)
string cpp_string = c_string;
```

- C++ string to C string

```
string cpp_string = "Bond Girls";
// char array must have enough space! Include '\0'!
char c_string[cpp_string.length()+1];
strcpy(c_string, cpp_string.c_str());
```


Data Structures and Algorithms
Estruturas de Dados e Algoritmos
2. Procedural Programming in C++
1 2 9 0 FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE D COIMBRA
DEPARTAMENTO DE ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

Input/Output from/to Files in C++

- Follows the same rules as with keyboard & console, both for *formatted data*, using >> and <<, including output formatting from `iomanip`, both for *unformatted data* using specific functions and methods to read/write characters or data bytes…
  - But the need to *declare a file descriptor*, *open file*, and *close file*…
- In C++, files are *also* I/O streams
- Which library needs to be included with `#include` ?
  - `#include <fstream>`

Example of Output to File in C++

```
#include <iostream>
#include <fstream>
using namespace std;          // to be explained later...
int main() {

  ofstream myfile;            // declare a file descriptor
  myfile.open("test.txt");    // open the file
  // check if the file's been open successfully
  if ( !myfile.is_open() ) {
    cout << "Could not open file for writing!" << endl;
    return -1;
  }

  double pi = 3.141592653589793;
  myfile << "PI is " << setw(5) << setprecision(3)
         << fixed << pi << endl;

  myfile.close(); // flush data and close the file

  return 0;
}
```

Very important to free resources used!

Data Structures and Algorithms
Estruturas de Dados e Algoritmos
2. Procedural Programming in C++
1290 FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE D COIMBRA
DEPARTAMENTO DE ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

File Opening

- By default, the file is opened in…
  - Text mode, for output, truncating any previous data

```
ofstream file("filename");
```

```
ofstream file;
file.open("filename");
```

- Modifying the default behavior:
  - `ofstream|ifstream|fstream <descriptor>(<filename>, <mode>);`

<mode>	Description
ios::out	Open file in write mode (for output)
ios::in	Open file in read mode (for input)
ios::trunc	Truncate any preexisting data in the file (if the file already exists)
ios::app	Append new data *at the end of file*
ios::ate	Go to the end of file on opening, without truncating any preexistent data, and allows adding new data at the end of file, or modify the existing data anywhere in the file
ios::binary	Open file in binary mode, i.e. for raw byte operation rather than character based

File Opening

```
ofstream file1("myfile.txt");
```
→ Open file for output with default options

```
fstream file1("myfile.txt", ios::out|ios::trunc);
```
→ idem…

```
ofstream file2("myfile.txt", ios::app);
```
→ Open file to append new text at the end

```
ofstream file3("myfile.txt", ios::app|ios::binary);
```
→ Open file to append binary data at the end

```
ifstream file4("myfile.txt");
```
→ Open file for input of text

```
fstream file5("myfile.txt", ios::in|ios::out);
```
→ Open file for input and output of text

```
fstream file6("myfile.txt",ios:in|ios::out|ios::binary);
```
→ Open file for input and output of binary data

Formatted Output/Input to Files

```
#include <iostream>
#include <iomanip>
#include <fstream>
using namespace std;

int main()
{
  double numbers[] =
    {1.0, -2.0, 2.4567, 3.141592653589793};
  int size = sizeof(numbers)/sizeof(double);
  ofstream file("numbers.txt");

  if (!file.is_open()){
    cerr << "Could not open file to write";
    return -1;
  }
  file << left << setw(6) << "Index" << right
       << setw(8) << "Data" << endl;
  for (int i = 0; i < size; i++)
   file << left << setw(6) << i << right <<
    <<setw(8) << setprecision(4) << fixed
    << numbers[i] << endl;
  file.close();
  return 0;
}
```

```
index   number
0       1.0000
1      -2.0000
2       2.4567
3       3.1416
```

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  const int size = 100;
  double numbers[size];
  ifstream file("numbers.txt");

  if (!file.is_open()){
    cerr << "Could not open file";
    return -1;
  }
  file.ignore(256, '\n'); //skip header
  int i = 0, j;
  while (file && i < size){
   file.ignore(6, ' '); //skip 1 column
   if (file) file >> numbers[i++];
  }
  file.close();
  return 0;
}
```

Reading & Writing Text to File
Unformatted Input/Output

- `get()` → read and return a character from file
- `get(char &c)` → read a character from file and store it in `c`
- `get(char *buffer, int compBuffer, char delim='\n')` → read `compBuffer` characters to a C string, or until `delim` is found, or until the end of file is reached
- `peek()` → read next character from file but keep it on file buffer
- `unget()` → restore to input file buffer the last character read
- `put(char c)` → write a character to file

Unformatted Text Input from File

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  // inicialize counters
  int cSpaces=0, cChars=0, cStatem=0;
  // open file
  ifstream file("Teste.txt");
  char c;

  if (!file.is_open()) {
     cout << "Error opening file"
          << endl;
     return -1;
  }

...
```

```
...
  // read file and count chars.
  while ( (c = file.get()) != EOF ) {
     switch (c) {
       case ' ': cSpaces++; break;
       case '\n':
       case '\t': break;
       case '.': cStatem ++; break;
       default: cChars++;
     }
  }
  // display results
  cout << "There are:" << endl
     << cSpaces << " spaces" << endl
     << cChars << " chars." << endl
     << cStatem << " statements"
     << endl;

  fich.close();
  return 0;
}
```

Reading & Writing Binary Data to File
Unformatted Input/Output

- Binary input/output:
  - Pros: no parsing effort, efficient storage, enables random access to data
  - Cons: not human-readable files, program more difficult to debug

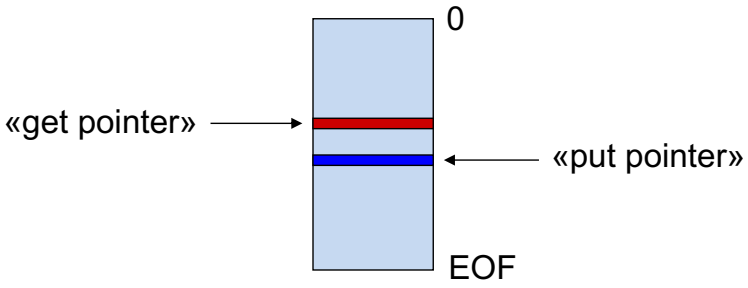
- `read(char *buffer, int n)` → read `n` data bytes from file and store data in array `buffer`
- `gcount()` → return the number of bytes read by previous call to `read()` method
- `write(char *buffer, int n)` → write to file `n` data bytes from array `buffer`

Unformatted Output/Input to Files

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  double numbers[] = {1.0, -2.0,
          2.4567, 3.141592653589793};

  ofstream file("numbers.bin",
                ios::binary);
  if (!file.is_open()){
    cerr << "Could not open file to write";
    return -1;
  }
  file.write((const char *) numbers,
             sizeof(numbers));
  file.close();
  return 0;
}
```

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  const int size = 100;
  double numbers[size];

  ifstream file("numbers.dat", ios::binary);
  if (!file.is_open()){
    cerr << "Could not open file to read";
    return -1;
  }
  int i = 0;
  while (file && i < size){
    file.read((char *) (numbers + i),
              sizeof(double));
    if (file.gcount() == sizeof(double)) ++i;
  } // i numbers have been read in the end
  file.close();
  ...
  return 0;
}
```

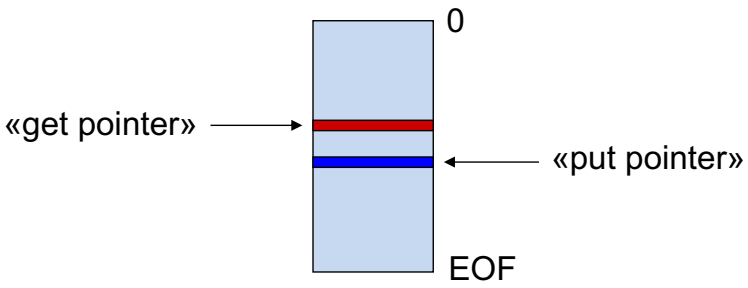
Binary Files – Random Access to Data



- `seekg/seekp(streampos pos)` → "seek get/put pointer", i.e. set the pointer to the *absolute* position `pos` w.r.t. the beginning of the file
- `seekg/seekp(streamoff offset, ios_base::seekdir dir)` → idem for the *relative* position `offset` according with `dir` :
  - `ios::beg` → `offset` with respect to the *beginning* of the file
  - `ios::cur` → `offset` with respect to the pointer's *current* position
  - `ios::end` → `offset` with respect to the *end* of the file

Data Structures and Algorithms
Estruturas de Dados e Algoritmos
2. Procedural Programming in C++
1 2 9 0 FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE D COIMBRA
DEPARTAMENTO DE ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

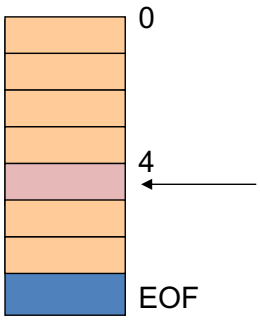
Binary Files – Random Access to Data



- `tellg()` → return the current position of the *get pointer*, i.e. the pointer for *reading* data
- `tellp()` → return the current position of the *put pointer*, i.e. the pointer for *writing* data

Binary Files – Random Access to Data

```
#include <fstream>
#include <iostream>
#include <cstring>

struct TProd{
  char code[10];
  double price;
};

int main() {
  Tprod prod;
  char code[10];
  double price;
  bool found;

  cout << "Insert the code of the
product to be modified: ";
  cin.getline(code,10);

  ...
```




```
  ...

  cout << "Insert its new price: ";
  cin >> price;

  fstream f("products.bin", ios::in |
            ios::out | ios::binary);

  if( !f.is_open() ) {
    cout << "Error: file not opened"
         << endl;
    return -1;
  }

  int idx = 0; // index of file record
  found = false;
  f.seekg(0, ios::beg);


  ...
```


Data Structures and Algorithms
Estruturas de Dados e Algoritmos
2. Procedural Programming in C++
1290 FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE D COIMBRA
DEPARTAMENTO DE ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

Binary Files – Random Access to Data

```
  ...

  while( !f.eof() ) {
    f.read ((char*) &prod,
            sizeof (TProd));
    if (f.gcount() == sizeof(TProd)){
      if (strcmp(prod.code, code)==0)
      {
        prod.price = price;
        f.seekp(idx * sizeof(TProd));
        f.write ((const char*) &prod,
                 sizeof (TProd));
        found = true;
        break;
       }
    }
    idx++;
  }

  ...
```

```
  ...

  if (!found) {
    cout << "Product " << code
         << " was not found!" < endl;

    strcpy(prod.code, code);
    prod.price = price;

    f.clear();  // reset flags
    f.seekp(0, ios::end);
    f.write ((const char*) &prod,
             sizeof(TProd));
  }

  f.close();
  return 0;
}
```

Dynamic Memory Allocation in C++

- Allocating memory – reserved word **new**

```
TProd *p1, *p3;      int *p2;

p1 = new TProd;      // allocate 1 data structure
p2 = new int[100];   // allocate array of 100 integers
p3 = new TProd[5];   // allocate array of 5 data structures

cout << "One price is " << p1->price << "." << endl;
cout << "Another is " << p3[4].price << "." << endl;
```

- Deallocating memory – reserved word **delete**

```
delete p1;
delete []p2;
delete []p3;
```


Data Structures and Algorithms
Estruturas de Dados e Algoritmos
2. Procedural Programming in C++
1 2 9 0 FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE D COIMBRA
DEPARTAMENTO DE ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

Dynamic Memory Allocation in C++

```
#include <iostream>
using namespace std;
int main() {
    int *grades;
    int i, numGrades;
    float average = 0.0;

    cout << "How many grades are you going to insert?"; cin >> numGrades;
    grades = new int[numGrades];

    if (grades != NULL) {
        for(i = 0; i < numGrades; i++) {
            cout << "Insert the grade " << i+1 << ":" << endl;
            cin >> grades[i];   average += grades[i];
        }
        average /= numGrades;
        cout << "The average grade is: " << average << endl;
    }
    ...
    delete []grades;
}
```

Reference Variable

- A *reference variable* represents an alternative name, i.e. an alias, for a variable
- A reference variable only makes sense if it is initialized when it is declared

```
int a;
int &ra = a;  // 'ra' refers to variable 'a'
ra = 2015;    //  this is equivalent to 'a = 2015'

int *pa = &ra;       // this is equivalent to 'pa = &a'
*pa = 2020;          // this is equivalent to 'a = 2020'

int &rb = ra; // 'rb' refers to the same as 'a' and 'ra'
```


Data Structures and Algorithms
Estruturas de Dados e Algoritmos
2. Procedural Programming in C++
1 2 9 0 FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE D COIMBRA
DEPARTAMENTO DE ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

Function Receiving Reference Variables

- Passing arguments by reference to a function

```
void writeDigits(int &num){
  while (num < 10)
        cout << num++;
}

int main( ) {
  int num = 2;

  writeDigits(num);
  cout << endl << "num="
       << num << endl;
}
```

23456789
num=10

Function Returning a Reference

- In C++, a function can return a reference
  - This allows the function appearing on the left side of an assignment!

```
int array[5]; // a global array variable
int invalid;

int& element(int index) {
        if( (index >= 0) && (index < 5) )
                return array[index];
        else return invalid;
}

...
element(2) = 5; // <=> array[2] = 5;
element(8) = 1; // <=> invalid = 1;
```


Data Structures and Algorithms
Estruturas de Dados e Algoritmos
2. Procedural Programming in C++
1 2 9 0 FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE D COIMBRA
DEPARTAMENTO DE ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

Function Returning a Reference

- But be careful with functions that return a reference!
  - Why doesn't the following function work correctly?

```
int& a_function(void) {
        int i = 50;
        return i;
}
```

Function Overloading

- In C++, multiple functions with the same name can be defined as long as they differ in the argument list
- This is denoted as *function overloading*

```
#include <iostream>
using namespace std;

double max(double a, double b);
int max(int a, int b);
double max(double array[], int size);
int max(int array[], int size);

int max(int a, int b) {
  return (a>=b?a:b);
}

...
```

```
...

int main () {
  int x = 3, y = 4;
  double z = 2.5, w = 1.3;
  int t[4] = {4, -1, 5, 3};

  cout << max(x, y) << endl; // 4
  cout << max(z, w) << endl; // 2.5
  cout << max(t, 4) << endl; // 5

  return 0;
}
```


Functions with Default Arguments

- Default arguments value can be specified in function declaration

```
int divide(int a, int b = 10) {
  int r;

  r = a / b;
  return r;
}
```

```
int main() {
  cout  << divide(60,3)
        << endl;
  cout << divide(16) << endl;
  return 0;
}
```

1. If an argument is initialized by default, *every* trailing argument also must be initialized
2. If an argument is omitted in the function call, *every* trailing argument must also be omitted
3. Expressions can be used to initialize an argument instead of constants, but these cannot refer other arguments or local variables of the function

Namespaces

- Only one program entity (variable, function, user-defined data type, etc.) can use a given label in a particular scope
  - This is seldom a problem for local labels, i.e. at the *block scope*
  - But name collision is likely to happen at the *global scope*, e.g. source file
- A *namespace* allows grouping named entities that otherwise would have global scope into a narrower scope: the *namespace scope*

```
namespace myNamespace {
  int a, b;
}
```

- The scope operator `::`

```
int a = 1, b = 2;
myNamespace::a = 3;
myNamespace::b = 4;

cout << a << ' ' << b
  << myNamespace::a << ' '
  << myNamespace::b
  << endl;
```


Namespaces

- Namespaces are particularly useful to avoid name collisions

```
#include <iostream>
using namespace std;

namespace foo {
  int value() { return 5; }
}

namespace bar {
  const double pi = 3.1416;
  double value() {
        return 2*pi;
  }
}
...
```

```
...

int main () {
  cout << foo::value() << endl;
  cout << bar::value() << endl;
  cout << bar::pi << endl;

  return 0;
}
```

```
5
6.2832
3.1416
```

Namespaces

- Keyword `using` ⟹ Now we finally understand `using namespace std;` !

```
namespace first {
  int x = 5;
  int y = 10;
}

namespace second {
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using first::x;
  using second::y;
  cout << x << endl << y << endl;
  cout << first::y << endl;
  cout << second::x << endl;
}
```

```
5
2.7183
10
3.1416
```

```
#include <iostream>
using namespace std;

namespace aSpace {
  int x = 5;
}

int main () {
  {
    using namespace aSpace;
    cout << x << endl;
  }

  return 0;
}
```

```
5
```